Correction Working with Java Data Types - 0

A - NullPointerException is thrown at runtime

Array elements are initialized to their default values. arr is referring to an array of Double type, which is reference type and hence both the array elements are initialized to null.

Correction Working with Java Data Types - 1

B - Exception is thrown at runtime

List cannot accept primitives, it can accept objects only. So, when 100 and 200 are added to the list, then auto-boxing feature converts these to wrapper objects of Integer type.

So, 4 items gets added to the list. One can expect the same behavior with remove method as well that 100 will be auto-boxed to Integer object.

But remove method is overloaded in List interface: remove(int) => Removes the element from the specified position in this list

and remove(Object) => Removes the first occurrence of the specified element from the list.

As remove(int) version is available, which perfectly matches with the call remove(100); hence compiler does not do auto-boxing in this case.

Correction Working with Java Data Types - 2

A - Point(10, 20);Point(0, 20)

HINT: First check if members are accessible or not. All the codes are in same file Test.java, and Point class & variable x, y are declared with default modifier hence these can be accessed within the same package. Class Test belongs to same package so no issues in accessing Point class and instance variables of Point class. Make use of pen and paper to draw the memory diagrams (heap and stack). It will be pretty quick to reach the result.

Point p1 = new Point(); means p1.x = 0 and p1.y = 0 as instance variable are initialized to respective zeros.

p1.x = 10; means replace 0 with 10 in p1.x,

p1.y = 20; means replace 0 with 20 in p1.y,

Point p2 = new Point(); means p2.x = 0 and p2.y = 0 as instance variable are initialized to respective zeros.

p2.assign(p1.x, p1.y); invokes the assign method, parameter variable x = 10 and y = 20.

As assign is invoked on p2 reference variable hence this and p2 refers to same Point object.

x = this.x; means assign 0 to parameter variable x, no changes in this.y, which means p2.x is unchanged.

this.y = y; means assign 20 to this.y, which means p2.y is now 20

So after assign method is invoked and control goes back to main method: p1.x = 10, p1.y = 20, p2.x = 0 and p2.y = 20.

Correction Working with Java Data Types - 3

D -

```
>
>0.0
>0.0
```

Primitive type instance variables are initialized to respective zeros (byte: 0, short: 0, int: 0, long: 0L, float: 0.0f, double: 0.0, boolean: false, char: 000).

When printed on the console; byte, short, int & long prints 0, float & double print 0.0, boolean prints false and char prints nothing or non-printable character (whitespace).

Correction Working with Java Data Types - 4

B - 2

Object created at Line 1 becomes eligible for Garbage collection after Line 1 only, as there are no references to it. So We have one object marked for GC.

Object created at Line 6 becomes unreachable after change(Pen) method pops out of the STACK, and this happens after Line 3.

Correction Working with Java Data Types - 5

E - [200, 100, 200]

List cannot accept primitives, it can accept objects only. So, when 100 and 200 are added to the list, then auto-boxing feature converts these to wrapper objects of Integer type.

So, 4 items gets added to the list: [100, 200, 100, 200]. list.remove(new Integer(100)); removes the first occurrence of 100 from the list, which means the 1st element of the list.

After removal list contains: [200, 100, 200].

NOTE: String class and all the wrapper classes override equals(Object) method, hence at the time of removal when another instance is passes[new Integer(100)], there is no issue in removing the matching item.

Correction Working with Java Data Types - 6

B - An exception is thrown at runtime

add(10.0, null); => Compiler can’t convert null to double primitive type, so 2nd argument is tagged to Double reference type.

So to match the method call, 10.0 is converted to Double object by auto-boxing and add(10.0, null); is tagged to add(Double, Double); method.

But at the time of execution, d2 is null so System.out.println(“Double version:” + (d1 + d2)); throws NullPointerException.

Correction Working with Java Data Types - 7

C - false

Boolean class code uses equalsIgnoreCase method to validate the passed String, so if passed String is “true” (‘t’, ‘r’, ‘u’ and ‘e’ can be in any case), then boolean value stored in Boolean object is true otherwise false.

In this question passed String is “ture” and not “true” and that is why false is printed on to the console.

Correction Working with Java Data Types - 8

A - Yes

Explication For readability purpose underscore (_) is used to separate numeric values. This is very useful in representing big numbers such as credit card numbers (1234_7654_9876_0987). long data can be suffixed by l, float by f and double by d. So first 5 variable declaration and assignment statements inside main(String []) method don’t cause any compilation error.

Let’s check rest of the statements:

l = c + i; => Left side variable ‘l’ is of long type and right side expression evaluates to an int value, which can easily be assigned to long type. No compilation error here.

f = c * l * i * f; => Left side variable ‘f’ is of float type and right side expression evaluates to float value, which can easily be assigned to float type. Hence, it compiles successfully.

f = l + i + c; => Left side variable ‘f’ is of float type and right side expression evaluates to long value, which can easily be assigned to float type. Hence, no issues here.

i = (int)d; => double can’t be assigned to int without explicit casting, right side expression `(int)d;` is casting double to int, so no issues.

f = (long)d; => double can’t be assigned to float without explicit casting, right side expression `(long)d;` is casting double to long, which can easily be assigned to float type. It compiles successfully.

Correction Working with Java Data Types - 9

C - JVM cannot be forced to run Garbage Collector.

Both Runtime.getRuntime().gc(); and System.gc(); do the same thing, these make a request to JVM to run Garbage Collector.

JVM makes the best effort to run Garbage Collector but nothing is guaranteed.

Setting the reference variable to null will make the object eligible for Garbage Collection, if there are no other references to this object. But this doesn’t force JVM to run the Garbage Collector. Garbage Collection cannot be forced.

Correction Working with Java Data Types - 10

B - true:false:false:null

Boolean class code uses equalsIgnoreCase method to validate the passed String, so if passed String is “true” (‘t’, ‘r’, ‘u’ and ‘e’ can be in any case), then boolean value stored in Boolean object is true otherwise false.

b1 stores true, b2 stores false, b3 stores false and as b4 is of reference type, hence it can store null as well.

Output is: true:false:false:null

Correction Working with Java Data Types - 11

A - null0.0false0

name, height, result and age are instance variables of Student class. And instance variables are initialized to their respective default values.

name is initialized to null, age to 0, result to false and height to 0.0.

Statement System.out.println(stud.name + stud.height + stud.result + stud.age); prints null0.0false0

Correction Working with Java Data Types - 12

D - Number version

There are 3 overloaded method m. Note all the numeric wrapper classes (Byte, Short, Integer, Long, Float and Double) extend from Number and Number extends from Object.

Compiler either does implicit casting or Wrapping but not both. 1 is int literal, Java compiler can’t implicit cast it to double and then box it to Double rather it boxes i to Integer and as Number is the immediate super class of Integer so Number version refers to Integer object.

Number version is printed on to the console.

Correction Working with Java Data Types - 13

E - NullPointerException is thrown at runtime

All the array elements are initialized to their default values. arr is of Boolean type (reference type), so arr[0] is initialized to null.

if expression works with Boolean type variable, so “if(arr[0])” doesn’t give compilation error but java runtime extracts the boolean value stored in arr[0] and it uses booleanValue() method.

arr[0].booleanValue() means booleanValue() method is invoked on null reference and hence NullPointerException is thrown at runtime.

Correction Working with Java Data Types - 14

A - None of the other options

switch can accept primitive types: byte, short, int, char; wrapper types: Byte, Short, Integer, Character; String and enums.

switch(b) causes compilation failure as b is of Boolean type.

Correction Working with Java Data Types - 15

B - After Line 3

At Line 3, p1 starts referring to the object referred by p2(Created at Line 2).

So, after Line 3, object created at Line 1 becomes unreachable and thus eligible for Garbage Collection.

Correction Working with Java Data Types - 16

A - 1

new Counter(); is invoked only once, hence only one Counter object is created in the memory.

c1, c2, c3 and counter are reference variables of Counter type and not Counter objects.

Correction Working with Java Data Types - 17

C - -108

127 + 21 = 148 = 00000000 00000000 00000000 10010100

Above binary number is +ve, as left most bit is 0.

Same binary number after type-casting to byte: 10010100, -ve number as left most bit is 1.

10010100 = -108.

Correction Working with Java Data Types - 18

A - null:null

Array elements are initialized to their default values.

arr is referring to an array of Boolean type, which is reference type and hence both the array elements are initialized to null and therefore in the output null:null is printed.

Correction Working with Java Data Types - 19

B - FTFFF

Boolean.valueOf(String s) returns true if passed String argument is not null and is equal, ignoring case, to the String “true”. In all other cases it returns false.

Boolean.valueOf(“abc”) => false. As “abc”.equalsIgnoreCase(“true”) is false.

Boolean.valueOf(“TrUe”) => true. As “TrUe”.equalsIgnoreCase(“true”) is true.

Boolean.valueOf(“false”) => false. As “false”.equalsIgnoreCase(“true”) is false.

Boolean.valueOf(null) => false. As passed argument is null.

Boolean.valueOf(“FALSE”) => false. As “FALSE”.equalsIgnoreCase(“true”) is false.

Correction Working with Java Data Types - 20

E - Compilation error in main method

extractInt method accepts argument of Double type.

extractInt(2.7); => 2.7 is double literal, so Java compiler would box it into Double type. At runtime obj.intValue() would print int portion of the Double data, which is 2.

extractInt(2); => Java compiler either does implicit casting or Wrapping but not both. 2 is int literal, Java compiler can’t implicit cast it to double and then box it to Double. So this statement causes compilation failure.

Correction Working with Java Data Types - 21

C - java.lang

All the wrapper classes are defined in java.lang package.

String and StringBuilder are also defined in java.lang package and that is why import statement is not required to use these classes.

Correction Working with Java Data Types - 22

B - FTFFF

Boolean.valueOf(String s) returns true if passed String argument is not null and is equal, ignoring case, to the String “true”. In all other cases it returns false.

Boolean.valueOf(“abc”) => false. As “abc”.equalsIgnoreCase(“true”) is false.

Boolean.valueOf(“TrUe”) => true. As “TrUe”.equalsIgnoreCase(“true”) is true.

Boolean.valueOf(“false”) => false. As “false”.equalsIgnoreCase(“true”) is false.

Boolean.valueOf(null) => false. As passed argument is null.

Boolean.valueOf(“FALSE”) => false. As “FALSE”.equalsIgnoreCase(“true”) is false.

Correction Working with Java Data Types - 23

E - Compilation error in main method

extractInt method accepts argument of Double type.

extractInt(2.7); => 2.7 is double literal, so Java compiler would box it into Double type. At runtime obj.intValue() would print int portion of the Double data, which is 2.

extractInt(2); => Java compiler either does implicit casting or Wrapping but not both. 2 is int literal, Java compiler can’t implicit cast it to double and then box it to Double. So this statement causes compilation failure.

Correction Working with Java Data Types - 24

C - java.lang

All the wrapper classes are defined in java.lang package.

String and StringBuilder are also defined in java.lang package and that is why import statement is not required to use these classes.

Correction Working with Java Data Types - 25

C - short s1 = 10;

E - final int i5 = 10; short s5 = 15 + 100;

G - final int i3 = 10; short s3 = 13;

Let’s check all the statements one by one:

short s1 = 10;

Above statement compiles successfully, even though 10 is an int literal (32 bits) and s1 is of short primitive type which can store only 16 bits of data.

Here java does some background task, if value of int literal can be easily fit to short primitive type (-32768 to 32767), then int literal is implicitly casted to short type.

So above statement is internally converted to:

short s1 = (short)10; short s2 = 32768;

It causes compilation failure as 32768 is out of range value.

final int i3 = 10; short s3 = i3;

Above code compiles successfully. If you are working with final variable and the value is within the range, then final variable is implicitly casted to target type, as in this case i3 is implicitly casted to short.

final int i4 = 40000; short s4 = i4;

It causes compilation failure as 40000 is out of range value.

final int i5 = 10; short s5 = i5 + 100;

Above code compiles successfully. If you are working with constant expression and the resultant value of the constant expression is within the range, then resultant value is implicitly casted. In this case, resultant value 110 is implicitly casted.

final int m = 25000; final int n = 25000; short s6 = m + n;

m + n is a constant expression but resultant value 50000 is out of range for short type, hence it causes compilation failure.

int i7 = 10; short s7 = i7;

Compilation error as i7 is non-final variable and hence cannot be implicitly casted to short type.

Correction Working with Java Data Types - 26

D - Three statements only

For readability purpose underscore (_) is used to separate numeric values. This is very useful in representing big numbers such as credit card numbers (1234_7654_9876_0987). Multiple underscores are also allowed within the digits. Hence, `int x = 5____0;` compiles successfully and variable x stores 50.

`float f = 123.76_86f;` compiles successfully.

1_2_3_4 is int literal 1234 and int can easily be assigned to double, hence `double d = 1_2_3_4;` compiles successfully.

____50 is a valid variable name, and as this variable is not available hence, int y = ____50; causes compilation error.

Underscores must be available within the digits. For the statement int z = 50____; as underscores are used after the digits, hence it causes compilation error.

Correction Working with Java Data Types - 27

A - Compilation error

In Java language, boolean type can store only two values: true and false and these values are not compatible with int type.

Also + operator is not defined for boolean types. Hence, all the 3 statements inside main method causes compilation error.

Correction Working with Java Data Types - 28

A - 97

Range of char data type is from 0 to 65535 and hence it can be easily assigned to int type. println() method is overloaded to accept char type and int type both. If char type value is passed, it prints char value and if int type value is passed, it prints int value.

As i1 is of int type, hence corresponding int value, which is 97, is printed on to the console.

Correction Working with Java Data Types - 29

C - Line n3 causes compilation error

Let us first check Line n1: byte b1 = 10;

Above statement compiles successfully, even though 10 is an int literal (32 bits) and b1 is of byte primitive type which can store only 8 bits of data.

Here java does some background task, if value of int literal can be easily fit to byte primitive type (-128 to 127), then int literal is implicitly casted to byte type.

So above statement is internally converted to:

byte b1 = (byte)10;

But if you specify any out of range value then it would not be allowed, e.g.

byte b = 128; // It would cause compilation failure as 128 is out of range value for byte type.

There is no issue with Line n2 as byte type (8 bits) can be easily assigned to int type (32 bits).

For line n3, `byte b2 = i1;`, expression on right hand side (i1) is neither a withing range literal value nor constant expression, hence it causes compilation failure.

To compile successfully, this expression needs to be explicitly casted, such as: `byte b2 = (byte)i1;`